Los Alamos
NATIONAL LABORATORY
EST. 1943

LA-UR-20-27295
Approved for public release; distribution is unlimited.

Title: Introduction to FLAG

Author(s): Whitley, Von Howard
DeBurgomaster, Carrie Ann

Intended for: Training and reference material.

Issued: 2020-09-17

Disclaimer:
Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by Triad National Security, LLC for the National Nuclear Security Administration of U.S. Department of Energy under contract 89233218CNA000001. By approving this article, the publisher recognizes that the U.S. Government retains nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

Introduction to FLAG

By Von H.Whitley, XTD-SS

Edited by Carrie A. DeBurgomaster, XTD-SS

The following guide is a tutorial-style introduction to FLAG for anyone who wants to get started with or learn more about FLAG. More information is available in the full FLAG manual.

Chapters 1-5 cover the basics of running FLAG. Chapter 5 is quite long, but covers a number of fundamental FLAG concepts, and is recommended reading. Chapters 6-8 lead the reader through simple HE cylinder and practical sample problems based on the Cagliostro experiments. After that, if you want to continue your FLAG education, Nick Denissen's Tutorial is a logical next step or you can read the FLAG manual (RTFM) found on the Lagrangian Applications Project (LAP) documentation page.

Confluence

The videos and files in this guide are linked to https://xcp-confluence.lanl.gov/. Employees who don't have access can contact their local group office to gain access.

High Performance Computing

You will need to log into a High Performance Computing (HPC) resource to run FLAG files.

Units

FLAG supports different units of measurement and the default is the centimeter-gram-microsecond (cgmu) system (pressure et al. follow):
Pressure: MBar
Density: g/cc
Velocity: cm/us
Temperature: K
HE Energy: MBar-cc/g
microseconds (μs)
gram (g)

XRAGE and users of other code users will become familiar with unit conversion.

Meshing

FLAG has many meshing options available - it can be run in Lagrangian mode, Eulerian mode, or some hybrid of the two. Since the Lagrangian meshes move, various mesh controllers are required to prevent mesh tangling. Depending on what you're running, mesh options can be fairly simple, or they can make up nearly half the input file. This will be discussed more later.

1. Setting up the Environment and Running FLAG

This section will cover setting up the environment to run an input file, and running a simple input to test that everything is working. There are two ways to run: interactively or batch script submission.

Running Interactively

Log into an appropriate yellow machine and ensure your FLAG file is on the machine. If you need to upload your FLAG file to the supercomputer:

- if you don't already have a specific location to run the model, create one on the supercomputer:
  cd /lustre/scratch3/yellow/*username*
  Note: there are other *scratch* locations besides *scratch3*. Files stored on the scratch space are cleared out regularly, so keep the 'master' file on your machine.
- in your *scratch* directory: mkdir *newdirname*
- cd *newdirname* to check that the directory was created
- change directories back to the area where your FLAG file is stored and secure copy the file:
  scp 2S-118.flg *username*@sn-fey:/lustre/scratch3/yellow/*username*/*newdirname*
  Note: the command above is for the Snow computer (sn=Snow).

Obtain an interactive allocation. Here is the command and output text requesting 1 node for 10 minutes:

```
[carried@sn-fey2 ~]$ salloc -N 1 -t 10
salloc: Pending job allocation 3562563
salloc: job 3562563 queued and waiting for resources
salloc: job 3562563 has been allocated resources
salloc: Granted job allocation 3562563
salloc: Waiting for resource configuration
salloc: Nodes sn215 are ready for job
This node is setup with a YELLOW environment.
[carried@sn215 ~]$
```

salloc is the command used to request the allocation.
-N is number of nodes
-t is time in minutes

When the allocation is granted and the prompt is returned, set up the location of FLAG executables. At the time of this writing, the most current version of FLAG was 3.8.Alpha.19. Running the most recent version is recommended.

```
setenv OPUS /usr/projects/shavano/SPUBLIC/releases/flag/3.8.Alpha.19/OPUS
setenv SUITE CTS1Ifast
source $OPUS/SHARED/config/$SUITE.modules
```

Now that the environment is set up, you can run an input file. To test this, download 2S-118.flg file from Confluence and execute the following:

```
srun -n 1 $OPUS/$SUITE/flag include 2S-118.flg |& tee flg.tty
```

Check that everything is set up and FLAG is running correctly. Input specifics aren't important yet. If the input is running correctly, you will start to see text scroll across the screen.

- The bulk will read **Ensight dump** followed by **Cycle**
  This means the input is running, it's dumping visualization files and cycling.
- At the end, there will be a comment: **Total Run time =**
- The information that scrolled across the screen while running can be found in the 'flg.tty' file.

Running as a Batch Script

If you would like to submit as a batch script, instead of running interactively, you will need to complete the steps in this section.

Edit your ~/.cshrc file using the editor of your choice. Then, add the following lines to add the location of the FLAG executable to the path:

```
setenv SHAVANO_HOME /usr/projects/shavano
set path=($SHAVANO_HOME/bin $path)
```

Save the file, then source in the .cshrc file (source .cshrc) or open a new window to get the updated path. If you did it correctly, you can type 'flag<tab>' at the prompt and see all the versions of FLAG that can be run. In this guide, we will run 3.8.Alpha.19.

```
flag.v_3.8.Alpha.19 -i 2S-118.flg -n 1 -w 10
```

In this case, we have requested:
-i is the input file name.
-n is the number of processors.
-w is time in minutes.

When the input starts to run, you should have a .tty file in the directory that contains all the run information. In the case of my run, the .tty file was called: 2S-118.flg.20200416_16_55_09.tty

Input File

Test environment setup: 2S-118.flg

2. Active Code Blocks, Comments, Variables, Math

Comments and Active Code Blocks

The comment sections are denoted by '$' or '!'. Both are interchangeable. So, in FLAG, that '$' means comment instead of active block. Additionally, in FLAG all blocks of code that do not begin with a '$' or a '!' are ***active***. You can also have a line that is partially active and partially a comment, such as:

```
/global/mesh/model/gas/eos $This is a comment embedded in an active line of code
```

Variables

FLAG can have variable declarations, which are most often used for setup dimensions or for defining a constant. For example, the 2S-118.flg file had the following variable definitions:

```
real Xlft, Xrht, TSTOP, FRight, ARight, VFLYER
integer Ncel,NV

 Xlft = 0.00 !(cm)
 Xrht = 5.00 !(cm)
 FRight = 1.0
 Ncel = 500

 TSTOP = 3.0
 VFLYER = 0.27983   $Velocity of the flyer (cm/us)
```

Those variables were used to quickly change the stop time, flyer velocity, or geometry.
TSTOP If you check the input, the stop time command (tstop) is set to the variable TSTOP.

Variable declarations are useful to many users. Understanding variable declarations makes .flg input files easier to work with.

Math

FLAG has the ability to do simple math in the input file. The input is a bit odd and goes by several names, including 'Polish Notation' and 'Prefix Notation'. If you're familiar with 'Reverse Polish Notation', it's the same thing, except the opposite -- Reverse Reverse Polish Notation! If you want to do something like 1 + 1 in this math, the FLAG input would be +(1 1). You will see this in input decks. Often, it will be a unit conversion or calculating a constant. Two examples are:

- RADIUS = *(5.0 2.54) $Convert a 5" radius to cm as in 5 * 2.54
- GAMMA = /(5 3) $Adiabatic gas gamma calculated as 5/3

You do not have to do math in the input deck, you can put in the constant value instead. However, sometimes it's useful for keeping original dimensions in inches (to make it easier to check input geometry and then calculate the correct dimensions in cm). You'll encounter such math operations in the sample input files provided in this tutorial.

3. Relational Database Input Structure; the "+" Shortcut

FLAG uses a relational database for the input format. A FLAG input deck will contain input such as:

```
mk /global/mesh/mat
```

Each one of the commands that follow a forward slash (/) are called "nodes", which has an unfortunate naming overlap with the nodes of a mesh (where mesh lines intersect). These are unrelated, so be aware of the double meaning of the term "node".

Each node can be considered a bucket of functions available in FLAG. The input manual follows this node structure, which makes it easy to find the location of a particular FLAG function in the input manual. When you open the manual, select the **collapsible version**:

> Hierarchy
> Original Version
> Collapsible Version

Let's say you want to take a look at the JWL function. That is found at:

```
/global/mesh/mat/gas/model/eos/jwles
```

Following the nodal hierarchy above will display something like this:

```
| . . . . bspall              BSpall:  Burton spall model.
| . . . . ejecta              Ejecta:  Ejecta model.
| . . . . eos                 GasEOS:  Directly use EOS class models.

| . . . . | bulkmod           Bulk:  Log Compression EOS.
| . . . . | dadjjwl           DadjJWL:  JWL HE model with Detonation Velocity adjustments.
| . . . . | davisprod         DavisProd:  Davis HE Products EOS  - 2nd eos for TwoEOS.
| . . . . | davisreact        DavisReact:  Davis HE Reactants EOS - 1st EOS for 2EOS.
| . . . . | eosbw             EosBW:  Prototype (?)  Default data type.
| . . . . | gamma             Gamma:  Gamma law gas EOS with optional stiffening.
| . . . . | gammahe           GammaHE:  Deprecated (?)  Gamma HE EOS.
| . . . . | gru2              Gru2:  Gruneisen EOS.
| . . . . | gruneisen         Gruneisen:  Gruneisen EOS.
| . . . . | jwl               JWL:  JWL EOS.
| . . . . | jwles             JWLES:  JWL EOS with energy shift.
```

Clicking on 'JWLES' will open information on the various parameters that go into the JWL. To use the JWL eos, variables under the JWL node need to be set. So, the input would look like:

```
mk /global/mesh/mat(HE)/gas/model/eos/jwles
    r0 = 1.834
    a  = 7.781315
    b  = 0.208618
    r1 = 4.50
    r2 = 1.50
    w  = 0.28
```

/mat(HE)/ is a tag on the mat node that is approximately the same as **material = HE**.

A different mat would receive a different tag. The Tags chapter (next) discusses tags in more depth.

Typing in all this relational structure can be very repetitive and time consuming. For example, the repetition for the full HE JWL block below:

```
mk /global/mesh/mat(HE)/gas
   region = "HE"

mk /global/mesh/mat(HE)/gas/element/hepoly
   detvelhe = 0.8825
   heenergy = 0.06434

mk /global/mesh/mat(HE)/gas/model/q/barton
   q2 = 2.0

mk /global/mesh/mat(HE)/gas/model/eos/jwles
    r0 = 1.834
    a  = 7.781315
    b  = 0.208618
    r1 = 4.50
    r2 = 1.50
    w  = 0.28

mk /global/mesh/mat(HE)/gas/initialize/ptre
   density = 1.834
   energy = 0.0
```

In addition to all the typing, if you want to reuse the code above for a material called mat(Booster), then all the mat(HE) nodes have to be changed. That requires 5 separate changes in the above code. In more complex sections of code, you might need to make 30-40 nodal changes! So much room for error.

Thankfully, there is a shortcut that can be used to help vastly reduce the amount of repetition. It's the use of '+'. The first time that '+' is invoked after a **mk** command, it inherits everything in the **mk** command.

This:

```
mk /global/mesh/mat(HE)/gas
        region = "HE"

mk /global/mesh/mat(HE)/gas/element/hepoly
        detvelhe = 0.8825
        heenergy = 0.06434
```

becomes much shorter using the + shortcut:

```
mk /global/mesh/mat(HE)/gas
   region = "HE"

mk +element/hepoly
   detvelhe = 0.8825
   heenergy = 0.06434
```

The '+' became shorthand for **mk /global/mesh/mat(HE)/gas**. It stays that same shorthand until it sees a new **mk** command that does not contain +. At that point, it resets. The full shortened version of the JWL block of code would look like:

```
mk /global/mesh/mat(HE)/gas
   region = "HE"

mk +element/hepoly
   detvelhe = 0.8825
   heenergy = 0.06434

mk +model/q/barton
   q2 = 2.0

mk +model/eos/jwles
    r0 = 1.834
    a  = 7.781315
    b  = 0.208618
    r1 = 4.50
    r2 = 1.50
    w  = 0.28

mk +initialize/ptre
   density = 1.834
   energy = 0.0

!-------------
!New material definition for copper
!-------------

mk /global/mesh/mat(Copper)/solid  !<------ This will reset what + stands for
.......
mk +element/fvpoly ! Now the + stands for /global/mesh/mat(Copper)/solid
```

This shortcut makes it easy to reuse blocks of code. If we wanted to reuse the JWL section above for a booster, we would need to change only the single **/mat(HE)/** to **mat(Booster)**. The rest of that block would then inherit **mat(Booster)**.

As you go about creating your own FLAG input, you can choose either approach, and you will see both notations mixed throughout local user models. This guide, going forward, will use the + notation as much as possible.

4. Tags

If you don't know how tags work, they can be really confusing. This section is strongly suggested reading.

In FLAG, it is common for some nodes to have multiple instances invoked. Examples:

boundaries : kbdy
regions : kregion
material : mat
mesh relaxers : volrelax

When you have multiple instances of a particular node, you have to call it something else. One obvious example is mat. If you have both an HE material and a booster material, you need to call them something different. So, we would have both:

.../mat(HE)/....
.../mat(Booster)/....

That's pretty straightforward. This means mat is a bucket of material names. In this bucket, there is an HE mat, a Booster mat, etc. Each one of the nodes listed above are separate name bucketspaces and are independent of each other. This can be confusing. It is quite common to see the following node names in an input deck:

/global/mesh/mat(HE)/.... Material block instance named HE
/global/mesh/kregion(HE)/.... Region block instance named HE
/global/mesh/kbdy(HE)/.... Boundary block instance named HE

Because they're different name containers, you can reuse the 'HE' tag multiple times. So, if you see a block of code like:

bdy = "HE"
material = "HE"

That first is referring to the HE boundary instance and is not calling either the material block instance or the region block instance. The second is calling a HE material instance and not the boundary instance. If you avoid using the same name for different nodes, it will minimize confusion. One suggestion for naming the above instances:

/global/mesh/mat(mat_HE)/....
/global/mesh/kregion(reg_HE)/....
/global/mesh/kbdy(bdy_HE)/....

Then the naming degeneracy is avoided when calling the boundary because

bdy = "bdy_HE"

looks completely different than a material call

material = "mat_HE"

5. A Simple Langrangian 2D Input Deck - PBX 9501 HE Cylinder

This section will detail and explain a full input deck (file linked at the end of this chapter). The previous chapters will make this fairly easy to understand. However, there are still a couple of things that we haven't discussed yet:

- function definitions
- defining problem boundaries
- defining and forming regions

The input is based off an experimental configuration. The geometry is simplified a little bit to make the setup easier; it should look like figure 1.

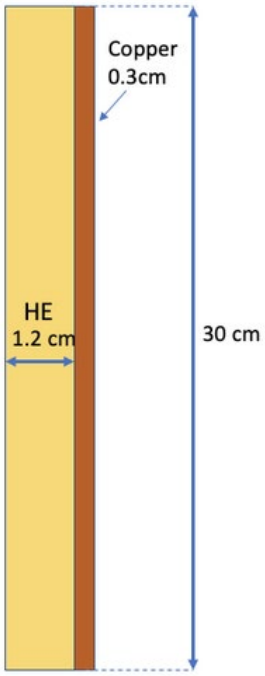


figure 1.

HE: The high explosive is PBX 9501 and has a radius of 1.2 cm.

Since we will be running in 2D cylindrical, the left edge of the HE will be an axis of symmetry.

Copper: The HE is surrounded by a 0.3 cm thick **copper** jacket.

PDV probes: Four PDV probes are included.

They are normal to the copper surface and are located at 7.875 in, 8.0 in, 8.938 in, and 9.0 in. Note those are in inches, not cm. The experimentalist used a standard ruler to measure them and sent me units in inches. I will use a math call to convert them to cm in the input deck. I do this because when we're trying to determine why the data and model do not match, the experimentalist will always ask if I put the probes in the right spot. I can quickly glance at the code and determine if they are in the correct position (in inches).

A small portion of the mesh we will be creating can be seen in figure 2.

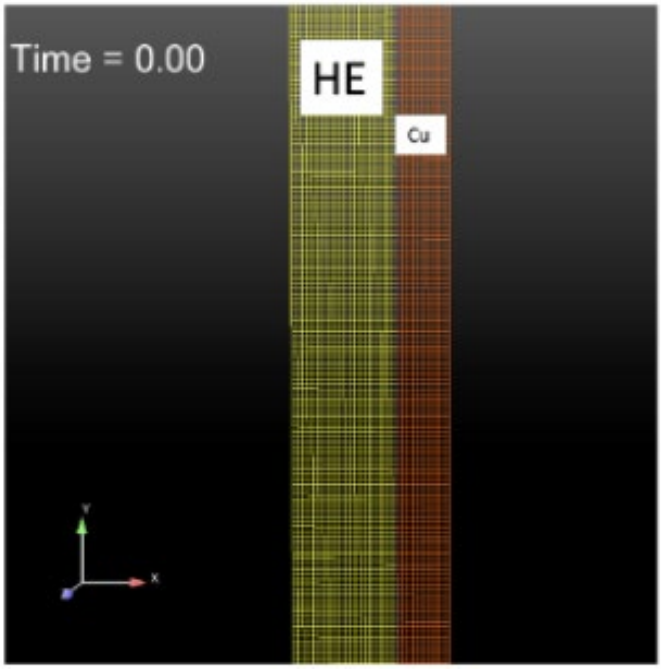


figure 2.

The cylinder extends off both the top and bottom of the image above. I had to zoom in on a section so the mesh was clearly visible. Note that the mesh covers only the HE and the copper cylinder. There is NO mesh outside of the copper! In an Eulerian code, mesh is needed outside the copper so that the copper can flow through the mesh. In a Lagrangian model like this one, the mesh moves with the material. With the mesh moving along with the copper, there is no need to add additional zones. However, when meshes are badly behaved at the outer boundary, we sometimes need to add additional zones outside the materials to help fix poor mesh behavior (detailed later in this chapter).

The FLAG Database Hierarchy will provide additional useful information for this section.

Problem Name and Stop Time

So, in the first few lines of input, we see:

```
mk /global
   title = "Lagrangian_Cylex"
   tstop = 45.0
   dtinitial = 1.0e-4
```

title is a prefix that code output is named.
In the case of this model, there will be a **Lagrangian_Cylex_input_echo** file (an echo of the input file) and a **Lagrangian_Cylex.visar** file that contains the PDV output.
tstop is the stop time of the model.
dtinitial is the initial timestep.

Meshing Options and Problem Geometry

The next portion of the code is associated with the symmetry and zoning the problem.

```
mk /global/mesh/geometry/axis2

mk /global/mesh/zoner/pszoner
   use_interval = "yes"

   ranges(1,:) = 0.0 1.2 1.5  !r dimensions
   izones(1,:) =    24  6     !Number of r zones (0.05 cm zones)

   ranges(2,:) = 0.0 30.0     !z dimensions
   izones(2,:) =    600       !Number of z zones (0.05 cm zones)
```

axis2 is the command for 2D cylindrical symmetry.

Under the **/geometry/** node there are other possible axes of symmetry.

pszoner is used to generate the mesh.

Refer to the database hierarchy to investigate other mesh types (this guide uses rectangular mesh as a default, but there are many other mesh types).

use_interv

This means FLAG will automatically decompose the problem and try to evenly spread the zones across the requested processors - so you don't need to make sure that the number of r zones and the number of z zones are evenly divisible by the number of processors.

ranges and izones specify the *r*- and *z*- lengths and the number of zones in them.

Define Functions

The next section of the code contains **func** commands. These are functions that evaluate true/false. If they're less than some value, they are false, if they're greater, they are true. These are odd to work with, in fact the manual entry states "*By themselves, these are useless*". The **func** commands in conjunction with other commands are used to create boundaries, regions of materials, etc. FLAG does not have simple graphics primitives. Things like 'box' and 'cylinder' that fill zones with a particular shape don't exist in a simple form. We will see how to use the func commands in later chapters to do things like fill in the copper walls of the cylinder. The block of code that defines the functions is:

```
mk /global/mesh/func(f_raxis)/planex   !Center axis of revolution location
   c = 0.0

mk /global/mesh/func(f_rmax)/planex   !Outer radial boundary location
   c = 1.5

mk /global/mesh/func(f_zmin)/planey   !Lower Z boundary location
   c = 0.0

mk /global/mesh/func(f_zmax)/planey  !Upper Z boundary location
   c = 30.0

mk /global/mesh/func(f_HE_Cu)/planex  !HE and Cu boundary
   c = 1.2
```

f_raxis is just a label for the function.

The 'f_' part of the name is used to delineate this name as a function. This function will be used later to define the axis of symmetry. What f_raxis is currently doing is defining a plane in which an r-value less than 0 is False. Any r-value greater than c=0 is true.

f_rmax will evaluate to False if the r-value is less than 1.5 and True if r-value is >= 1.5.

Visually, the two r value functions look like figure 3.

f_z_min and **f_z_max**

- use a 'planey' to define Z boundary locations
- use cylindrical symmetry, colloquially called R-Z
- In FLAG, the r-direction maps to the x-coordinates, and the z-direction maps to the y-coordinates. In FLAG cylindrical, the z component is not used; FLAG will crash if a z component is entered.
- The result is a slightly confusing function definition in cylindrical coordinates; the z boundaries defined by planey calls. These func calls tend to be more or less boilerplate, where the c-values get changed to match the outer bounds of the problem. So, never changing that block of code will simplify functions. Visually, the two Z boundary functions look like figure 4.

f_HE_Cu is used to define the location between the HE and the inner copper wall.

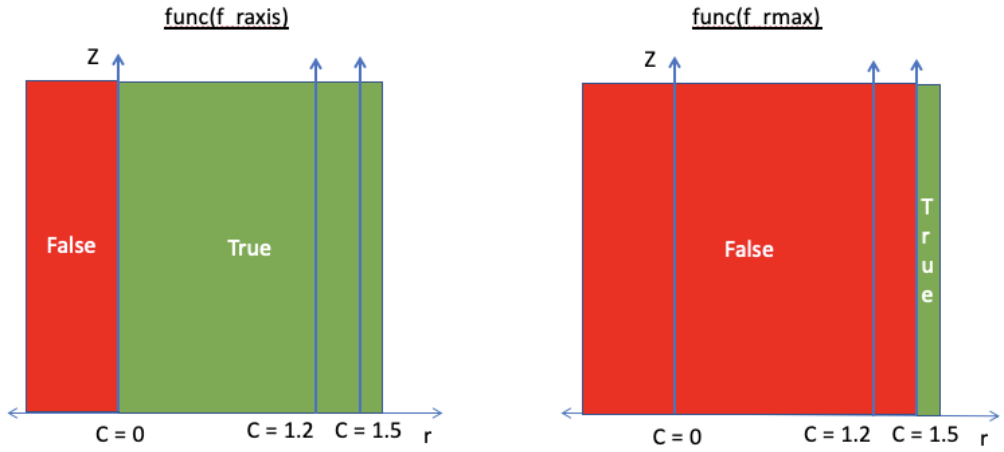


figure 3.

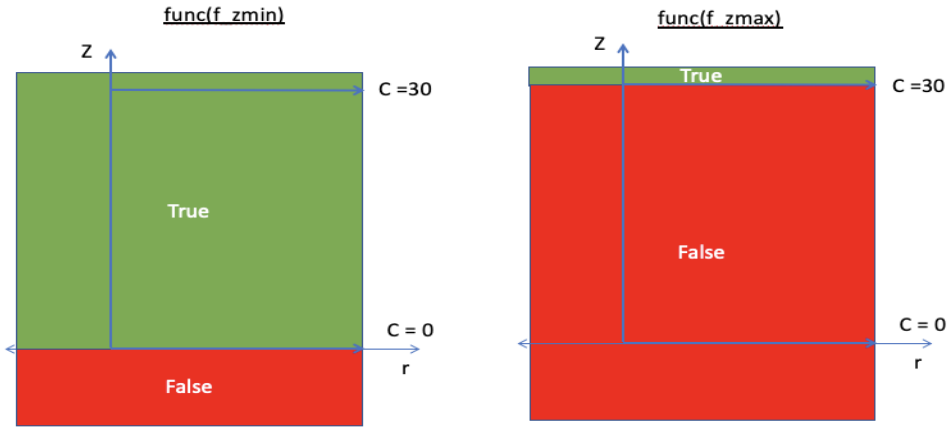


figure 4.

Define Boundaries

Now that we know what the functions are doing, we can use them to create the boundaries of the problem.

```
mk /global/mesh/kbdy(bdy_raxis)/onefunc
   fname = "f_raxis"

mk /global/mesh/kbdy(bdy_rmax)/onefunc
   fname = "f_rmax"

mk /global/mesh/kbdy(bdy_zmin)/onefunc
   fname = "f_zmin"

mk /global/mesh/kbdy(bdy_zmax)/onefunc
   fname = "f_zmax"
```

/kbdy/onefunc function defines a boundary (a set of points) using the 'c =' values of the func calls.

- **kbdy(bdy_raxis)/onefunc** defines the r axis boundary.

fname = "f_raxis" command translates as:

Create a boundary set of points at the c value of the func "f_raxis". In this case c=0, so create a boundary set of points at c=0, then call this boundary "bdy_raxis".

The remaining three **kbdy** commands do similar things except at the different respective **func** calls.

At the moment, these newly created boundaries do nothing, they are merely a set of points that happen to lie on the edge of the problem. These boundaries do not know how to behave until directed, for example: 'you can only move in the r-direction' or 'you cannot move anywhere'. This is done through the boundary condition function **/hydro/lhydro/kbc/kfix**. A variable called **nfix** under the **kfix** node actually tells the boundary how the points can move.

figure 5 shows what the different nfix conditions look like for a set of points. **nfix = rdir zdir** where 0 allows motion in a direction and 1 fixes/restricts the ability to move in a particular direction.

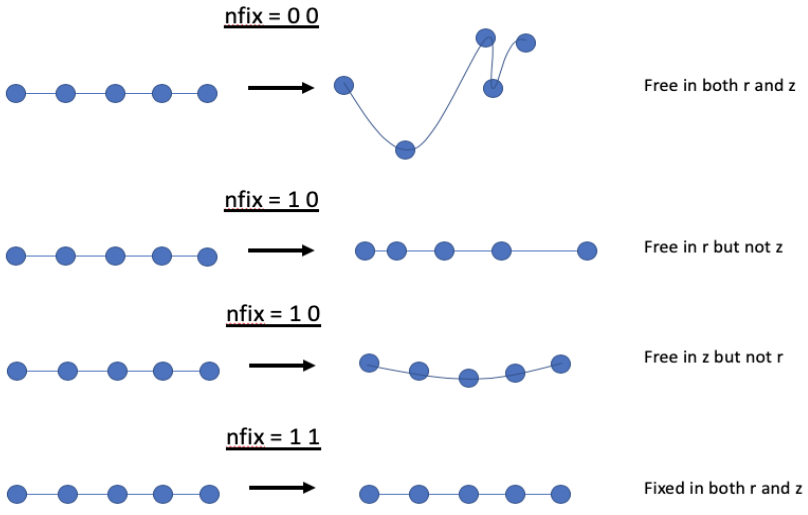


figure 5.

Note: the boundary conditions can have a **tstart** and **tstop** associated with them. That means at a given model time, one type of boundary condition can be stopped and another invoked, which can be useful for controlling bad mesh behavior.

Looking at the block of code containing the mesh boundary conditions, we see:

```
mk /global/mesh/hydro/lhydro
   alias pressure zp
   alias density zr
   alias velocity pu

mk +kbc(bc_zmin)/kfix
   bdy = "bdy_zmin"
   nfix = 0 1

mk +kbc(bc_zmax)/kfix
   bdy = "bdy_zmax"
   nfix = 0 1

mk +kbc(bc_raxis)/kfix
   bdy = "bdy_raxis"
   nfix = 1 0
```

* click here for info on '+'

lhydro is used to run the Lagrangian hydro; it's in every input deck. Even when running FLAG in 'Eulerian' mode, lhydro moves the mesh before it is remapped back to its original position.

alias - under hydro there are 3 aliases, which are material conditions. Save for export to Ensight (dumps). The name of the variable in the code is zp. It is easier to understand variables in Ensight when they are aliased. In the example above, the variable zp is aliased to the name "pressure", which will cause the variable to be called "pressure" instead of "zp" in Ensight.

mk +kbc are the boundary conditions to invoke for the edges of the mesh.

- **raxis**: Allow the nodes along the r-axis to be fixed in r but move in z, which is the desired behavior for the axis of symmetry. This will prevent the r axis from pulling off 0 and moving to a positive r, or worse, becoming a negative r.
- **zmin**: The nodes along the z min boundary are fixed in z but can move in r. Normally, we expect the HE gases to expand out of the copper cylinder, and this rapid mesh expansion will quickly start to tangle. To control the expansion, add a condition that does not allow the gases to expand in the -z direction. This will make the model run without issues, but the first inch or so of cylinder data will probably not be a good comparison to experimental data. Just make sure PDVs are located away from the zmin boundary.
- **zmax** is similarly fixed so that the gases cannot expand out of the top of the cylinder.

Another choice is whether to trade off model fidelity for mesh robustness. The final two commands just after the boundary conditions are boilerplate. They are commonly used, but not commonly understood.

```
mk +mixedcell/mxtip

mk /global/mesh/mat(mat_mqtts)/modq/mqtts
   region = "universe"
   q1 = 0.1
   q1n = 0.1
   q2 = 1.3
```

mxtip is a command which determines how mixed cells are compressed.

The mxtip option means that lower density materials in a mixed zone undergo larger compressions compared to higher density components.

mqtts is a mesh stiffener command to help with thin zones.

Sometimes zones will ‘chevron' or 'bowtie'. In this case, the corner (or opposite corners) can invert, which can cause major problems. Imagine a zone corner is your knee. Chevroning is the equivalent of your knee bending backward. Not good.

Defining Regions

The next block of code will show how to create regions. Namely, they're going to be the geometric regions to assign material parameters to. Note that **func** is used again, but this time in conjunction with the region creation function **kregion.**

```
mk /global/mesh/kregion(universe)/universe

mk /global/mesh/kregion(reg_Cu)/onefunc
   fname = "f_HE_Cu"

mk /global/mesh/kregion(reg_HE)/boolregion
   kbool = Expr(comp(reg_Cu))
```

/kregion/universe creates a region that encompasses the entire problem. Create this first.

The universe region (region = “universe”) can be called in order to do something to the whole mesh, such as stiffen it.

f_HE_Cu uses a function to create a region.

Here, the func is placed at the boundary of the HE/Copper. Recall that the function evaluates to True from >= 1.2 cm, and that copper occupies the entire mesh beyond 1.2cm. So, the copper region is created with this one call.

reg_He creates an HE region.

This region requires a function that evaluates to true from r= 0 to r= 1.2 cm. The **func** commands don't have that behavior. They're true when greater than a c value, but never become false again. A function called **boolregion** can resolve this issue. This function can do things like union operations, intersection operations, and complement operations. A complement of the **f_HE_Cu** function has the desired behavior. So, the final command above uses the complement operator to create our **reg_He** region.

Copper Material Definition

The next step is to define the materials. There are two: copper and explosive. Copper will be defined as a **/mat/solid**. The strength and damage functions all lie under the solid node, so anything with strength needs to be a solid. The HE will be defined as a **/mat/gas**. All the reactive burn and program burn functionality are under the gas node.

The material definition for Copper using Sesame 3336 and a PTW damage model looks like:

```
mk /global/mesh/mat(mat_Cu)/solid
   region = "reg_Cu"

mk +element/fvpoly

mk +model/q/barton
   q1 = 0.3
   q2 = 1.33

mk +model/decoupled/pvol/eos/sesame6re
    ideos = 3336

mk +model/decoupled/strength/ptw
    r = 0.                !Rate smoothing parameter.  NOT DENSITY!
    theta0=0.025
    p=2.0
    kappa=0.11
    gamma=1.0e-5
    alpha=0.2
    g0=0.518               ! Mbar
    tm=1356                ! deg-K
    am=1.0552e-22          ! g/atom
    s0=0.0085
    sinf=0.00055
    y0=0.0001
    y1=0.094
    y2=0.575
    yinf=0.0001
    beta=0.25

mk +model/decoupled/strength/ptw/meltt/eospac6
    ideos = 33336

mk +model/decoupled/strength/ptw/shearm/eospac6
    ideos = 33330       !There is no 33336 shear table.

mk +initialize/stre
   density = 8.93
   energy = 0.0
```

The material definition text is mostly self explanatory:

/q/barton command is used to set the artificial viscosity (AV) to a metal specific version.

FLAG can define the artificial viscosity on each material. Often, a different artificial viscosity is used for metals compared to HE.

/element command sets the elements to a finite volume polyhedral.

fvpoly or **hepoly** are for materials. If you're not program burning something, it will be **fvpoly.**

/initialize/stre to initialize the material.

Initializes as a function of stress, temperature, density, energy and the deviatoric stress tensor. Provide two of the four conditions and FLAG will solve for the other two.

PBX 9501 Material Definition

The material block for PBX 9501 (using Lund program burn) looks like:

```
mk /global/mesh/mat(mat_HE)/gas
   region = "reg_HE"

mk +element/hepoly
   detvelhe = 0.8825
   heenergy = 0.06434

mk +model/q/barton
   q1 = 0
   q2 = 2.0

! PBX 9501 parameters
mk +model/eos/jwles
    r0 = 1.834
    a  = 7.781315
    b  = 0.208618
    r1 = 4.50
    r2 = 1.50
    w  = 0.28

mk +initialize/ptre
   density = 1.834
   energy = 0.0
```

/mat/gas is the explosive defined as a mat/gas.
hepoly is the elements defined as hepoly.

The only materials that are ever hepoly are program burns.

If you're using Lund or DSD, the HE will be a **hepoly**, everything else will be **fvpoly**.

barton is where the gas-specific artificial viscosity is used.
ptre is used to initialize.

Since gas materials do not have deviatoric stress tensors, we cannot use the **stre** initialization found in solid. So, gases use **ptre** initialization, which is roughly equivalent.

Lund Program Burn

The next portion of the code defines the **lund** program burn input. It's a simple burn model (burnform = 'prog'), and is analogous to the $Dets section.

```
mk /global/mesh/heburn/helund

mk /global/mesh/heburn/hedet
   !     X    Y   t
   dxt = 0.0 0.0 0.0
```

dxt is pretty straightforward: start the initiation at X = 0.0 and Y = 0.0 at time t= 0.0.

Keep in mind that r-z geometry in FLAG is represented by x-y and not x-z. Do ***not*** include a coordinate for the z light point, it will cause FLAG to crash - unless you are running a full 3D model. The database hierarchy has more **hedet** information, such as other light options like line light, etc.

PDV Definitions

FLAG defines the PDV by origin point and a direction that is a unit vector composed of the x-component and the y-component. Here, these PDVs point normal to the copper surface and are located at 7.875", 8", 8.938" and 9". I use FLAG math to convert from inches to centimeters. Note that the PDV are located outside the problem at r = 14.3 cm and they all point inward along the -r direction. A couple of things need to be specified for PDV:

```
mk /global/mesh/output/visar_pdv
   bdy = "bdy_rmax"
   matlist = "mat_Cu"
   kkvll = 4
   vorigin = 14.3       0.0     *(7.875 2.54) &
             14.3       0.0     *(8.0 2.54) &
             14.3       0.0     *(8.938 2.54) &
             14.3       0.0     *(9.0 2.54)
   vdir    = -1.0       0.0     0.0 &
             -1.0       0.0     0.0 &
             -1.0       0.0     0.0 &
             -1.0       0.0     0.0

   doc BuffVisarPDV every 10
   dot VisarPDVDump every 1.0
```

bdy = "bdy_rmax" means we're going to monitor the rmax boundary.
We need to specify the boundary that the PDV will be intersecting.
matlist = "mat_Cu" specifies the material we want to monitor.
kkvll = 4 means that we are using 4 PDV probes.
& is used at the end of the pdv coordinates to continue the input.

Finally, there are two commands that deal with PDV buffering and writing:
doc stands for “Do On Cycle” and will buffer the PDV every 10 cycles.
dot stands for “Do On Time” and will write the pdv dump file every 1.0μs.
Note that these doc and dot commands do not need to be located with the PDV. There is a time controller section of the code that controls time for other parts of the problem; co-locating all the time controllers there would be another way of organizing the code.

Ensight Output

This block of code controls the output dumps for ensight.

```
mk /global/mesh/output/ensight
    filepath = "./ensight"
    defsets  = "mats"
    nvars = 3
    vars =  "pressure" &
            "density" &
            "velocity"
    iensmatint = 1 $Plot material interfaces
    iensvisar = 1  $Plot PDV probe rays
```

filepath is the location of the dumps. FLAG will create the directory if it does not exist.
defsets defines a set of common variables that will be saved.

“mats” means that every real material in the problem will dump:
hr (density), *he* (specific internal energy), *ht* (temperature), *hp* (pressure), and *hhv* (volume).
nvars specifies we are saving 3 aliased variables: "pressure", "density" and "velocity".
Those 3 variables are the ones we aliased under the **lhydro** node. Note that the "pressure" and "density" variables are semi-redundant with the set specified by defsets = "mats".

Time Controllers

This section of input controls two additional time controllers for **DTC** and **ENSIGHT**

```
doc DTC every 10
dot ENSIGHT every 0.5
```

DTC is used to report the cycle information.
doc (Do On Cycle) is used to dump the cycle information that every 10 cycles.
dot (Do On Time) is used to dump Ensight dumps every 0.5 µs.

Problem Execution

```
run
end
```

The last two commands tell FLAG to **run** the input and to **end** the execution when a stop condition has been met. Pretty much every input deck ends with these two commands, consider it boilerplate. There are some advanced run control options to interrupt the run and interactively do stuff in the code, but for our purposes, they are advanced/developer features.

To continue with the steps in this guide, you need to download and run the "SimpleCylinder.flg" input deck at the end of this chapter. Chapter 6 uses the dump files with Ensight to take a look at the model results. figure 6 shows how your Cylex model should look at 13.5 µs when plotting 'pressure' in Ensight. Units are in MBar.

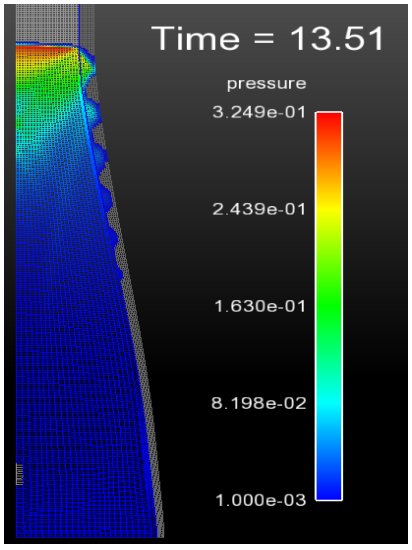


figure 6.

Input File

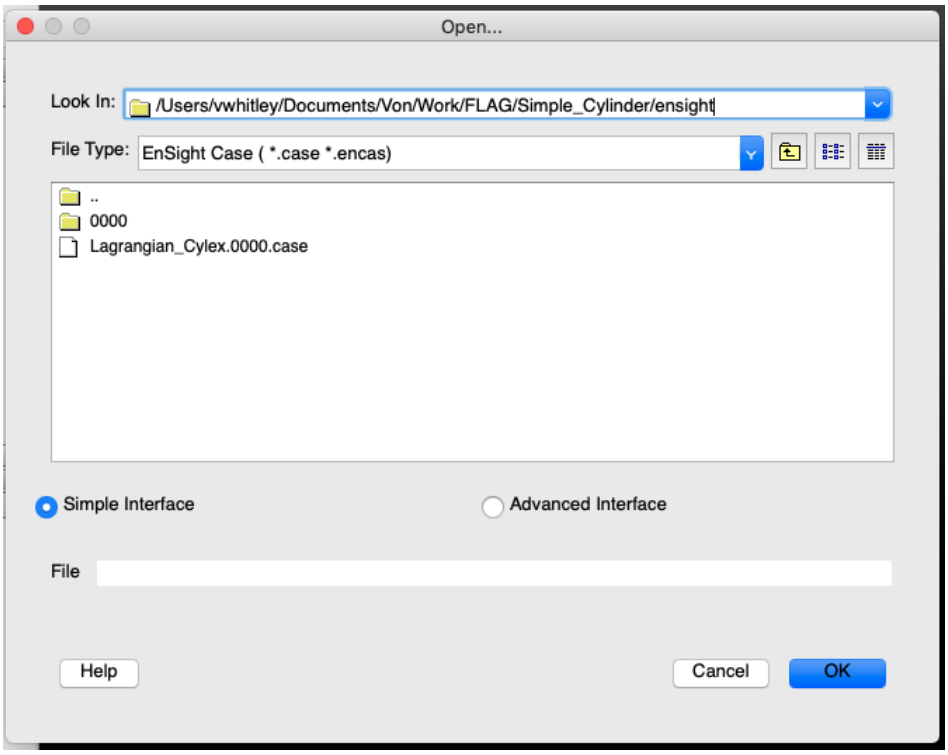
Part 5 input file: SimpleCylinder.flg

6. Simple HE Cylinder Problem - Using Ensight to Display Results

In this chapter, we will cover plotting FLAG dumps/results in Ensight. This assumes you are already familiar with Ensight.

Loading the 'Case'

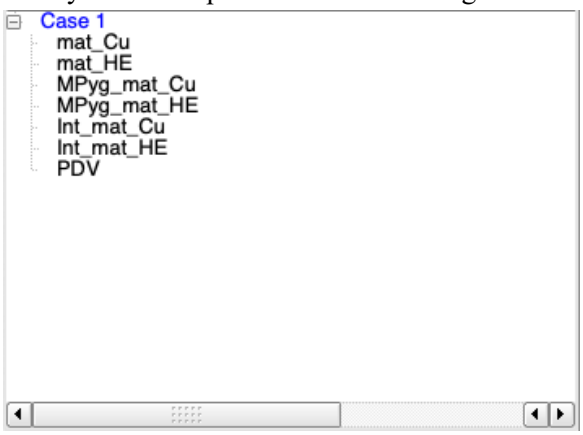
Use HE cylinder model results or the results linked at the end of this chapter. To view the results of the model, first load the 'case' file into Ensight: start up Ensight, go into the '/Ensight' directory, and load the case file.



Note that there is only a single case file that contains all of the information. All of those results are found in the /0000 folder. Opening up the Ensight directory in FLAG is considerably faster because it isn't listing all the dump files.

Parts Loaded by the Case

Here you see the parts loaded into Ensight.



mat_Cu and **mat_HE** are our two materials defined in the .flg deck.
Mpyg are mixed zones.

Mixed zones typically occur when ALE is invoked to fix a bad mesh. This problem ran fully Lagrangian, so there are no mixed zones.

Int_mat_HE is the material interface for the HE, which is the interface between the HE and Cu.
Int_mat_Cu is the interface located along the outer Cu boundary.

Since there is no other material there, this interface doesn't show anything.

Producing a Pressure Plot

Click on the materials to plot. Select all, or to see the pressure of a single material, just select one (to see copper, select mat_Cu). Proceed to plot the pressure variable. In the case below: right-click on the case, select 'Color by', then 'Select Variable', and finally 'pressure'.

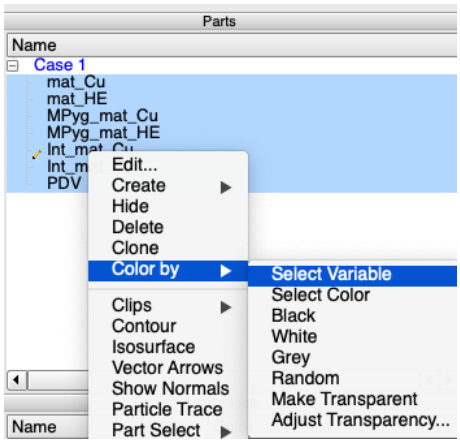


Figure 7 shows the resulting plot (when time is 13.51 us, frame 27 in my model).
Your results should be similar.

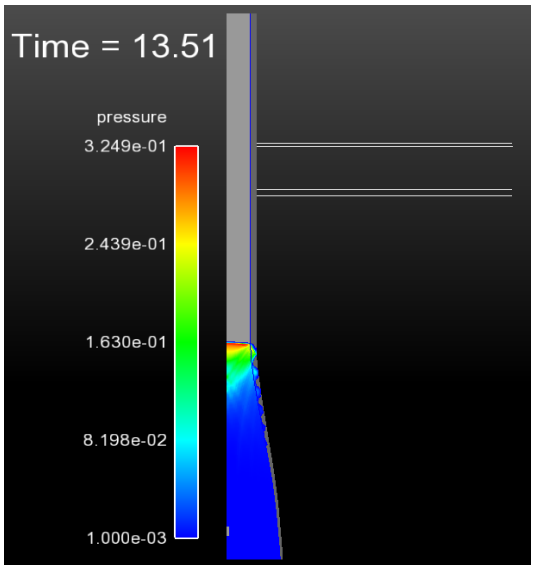


figure 7.

The 4 horizontal lines on the outside of the copper are the PDV probes. We specified that we wanted to plot those in the Ensight node of the model.

Dump Files

Dump files for the HE cylinder model of Part 5: SimpleCylinder_Ensight.tar.gz

7. Using Ingen to Create the Cagliostro Mesh

The input for the previous cylinder model used the built-in meshing functionality of FLAG to generate the mesh. Most meshes used with FLAG are not generated this way. Almost all of them are generated using a different piece of software called Ingen. This chapter will not be about FLAG – it will cover the process of making a relatively simple mesh using Ingen.

Mesh commands themselves are written in Python, then Ingen takes those commands and creates the mesh. Once this mesh is generated, it can be imported into FLAG and used for a model. The next chapter will cover how to import the mesh and run it in FLAG.

To begin, load the modules necessary to access Ingen. Log into a yellow machine (ex: Snow). When prompted, enter:

```
module use /usr/projects/setup/modules
module load setup
```

Ingen will load. Start a new Python text file for creating the mesh. The file is linked at the end of this chapter, for those who prefer to download it instead of creating it. The mesh we are going to create is based on some experiments by Dominic Cagliostro[1].

Cagliostro's base experiment was a hemisphere of PBX 9501 driving an inner hemispherical metal liner. He tested two different metal liners: copper and tantalum. Each experiment had 2 VISAR probes (an early predecessor of PDV), shown in figure 8.

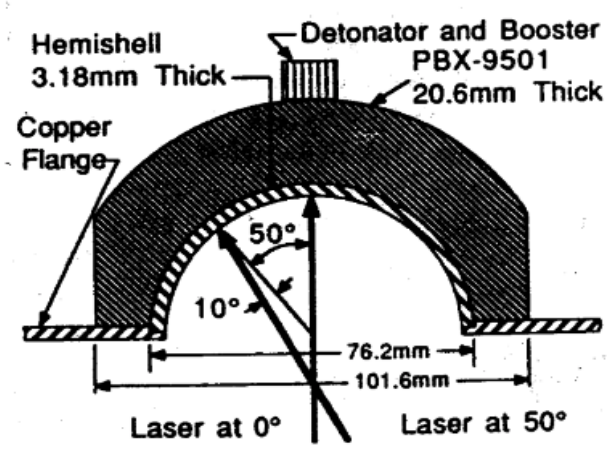


figure 8.

[1] D. Cagliostro, R. Warnes, N. Johnson, and R. Fujita, "Spall Measurements in Shock-Loaded Hemispherical Shells From Free-Surface Velocity Histories," *Shock Waves in Condensed Matter*, LA-UR-87-2292, 1987

Figure 9 shows what the mesh will look like in the simplified design used in this chapter. The detonator/booster and copper flange are left out, and the outer HE contour is spherical. A later section will cover these additional geometric complications.

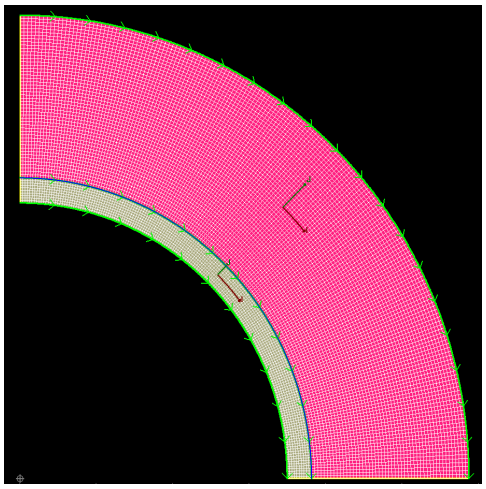


figure 9.

The geometry simplifications make the meshing file short and simple. The rest of this chapter will discuss the various functions needed to generate the mesh. The Ingen Manual details the functions that we are using and shows examples of other functions available. An easy way to navigate it is to type in the function in the search box in the upper right. For example, if I want to know more about gwiz.arc and to find similar geometry commands, I type in "arc" in the search box (if you type "gwiz.arc", it won't find anything). A drop down box of 'arc' functions will appear. Click on the gwiz.arc and it will take you to the right section.

Back to the details in the Cagliostro_mesh.py file:

```
import ingen
from ingen import gwiz, altair, materials

ingen.startModel(name='mesh/Cagliostro',convenience=globals())
```

import ingen Python import statement needed to access the functionality of ingen.
from ingen import gwiz, altair, materials Python import statements needed to access modules.
Specifically, models for geometry building and meshing.
startModel is a required command, the name will be the mesh filename.

Next, there are 4 python variables used to set the dimensions of the geometry. The dimensions are the experimental dimensions from figure 9 (above). This defines the geometry and makes it easy to change.

```
inner_he_diameter = 3.81
metal_thickness = .318
he_thickness = 2.06
res = 0.5
```

Start the meshing process by making lines, arcs, ellipses, and tabular contours that define all interfaces in the problem. The three interfaces in the simplified Cagliostro geometry are all arcs. Since RZ symmetry will be used in the FLAG model, we only need to define the arcs from 0 to 90 degrees. The three interface contours are created with:

```
cntr.inner_metal = gwiz.arc(radius = (inner_he_diameter - metal_thickness), startAngle = 0, endAngle = 90)
cntr.metal_he_bdy = gwiz.arc(radius = inner_he_diameter, startAngle = 0, endAngle = 90)
cntr.outer_he = gwiz.arc(radius = (inner_he_diameter + he_thickness), startAngle = 0, endAngle = 90)
```

This creates contours at the various radii listed in figure 8, the experimental diagram above. The three gwiz.arc commands above generate the three arcs in figure 10 below.

Alternately, to generate these contours yourself: open Cagliostro_Mesh.py, comment out all lines after the cntr section except for the **ingen.endModel**, and run the script using *ingen -g Cagliostro_Mesh.py* at the prompt. That should generate the graphic in figure 10.

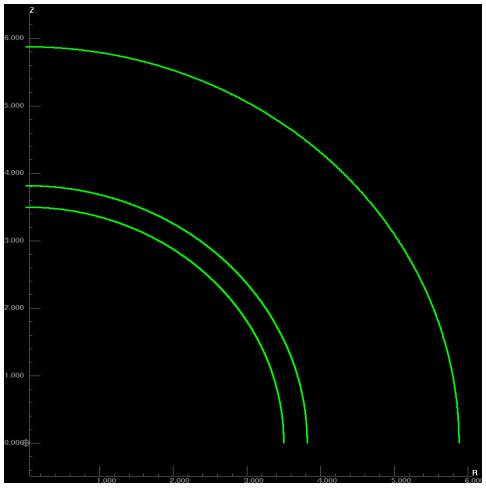


figure 10.

These contours need to be segmented in order to make the mesh out of the segments.

```
altair.contours2Segments(contours=cntr,segments=seg)

seg.inner_metal.equalAngleDistrib(dAngle = res)
seg.metal_he_bdy.equalAngleDistrib(dAngle = res)
seg.outer_he.equalAngleDistrib(dAngle = res)
```

contour2segments creates segments on each contour by specifying the spacing between segments.

- **contours=cntr** is used because all current contours lie in the cntr namelist.
- **segments=seg** specifies that all segments lie in the seg namelist.

dAngle = res sets the angular spacing. It is the only segmentation option we set.

This handles spacing between segments and ensures the number of segments on the inner contour are the same as the number of segments on the outer contour. The number of zones will be the same, but the ones along the outer part of the problem will be fatter.

Figure 11 shows what the segmented contours will look like:

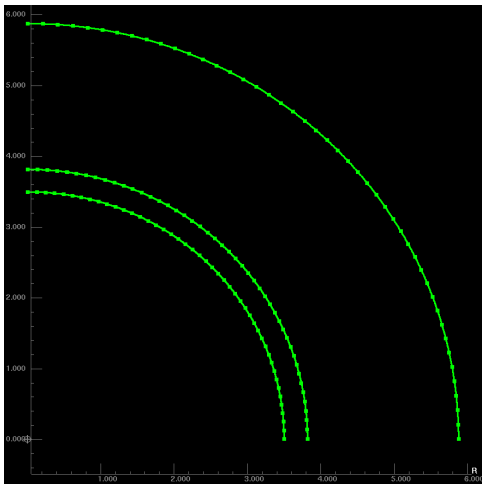
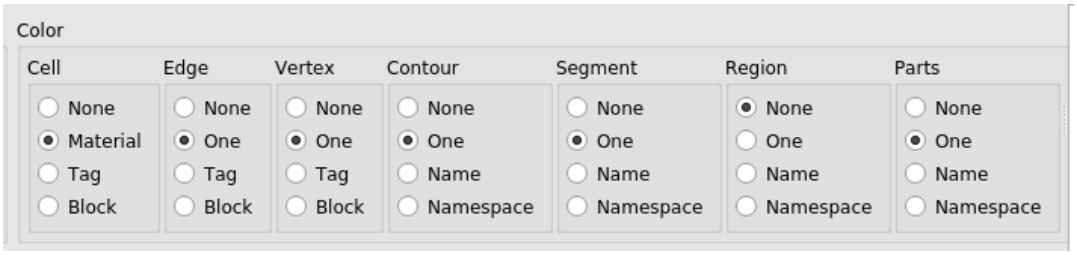


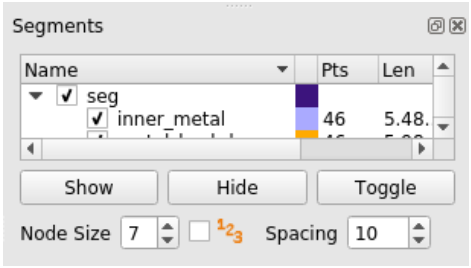
figure 11.

To generate these contours yourself: open the Cagliostro_Mesh.py file, comment out everything after the last seg command except the ingen.endModel, and run the script using *ingen -g Cagliostro_Mesh.py* at the prompt. This should generate the graphic in figure 11.

Note: The 'res' variable is increased to 2° in order to clearly show the different segments. At 0.5°, they overlap. Recommended display options below:



Check the Segment Node Size in the right tool bar to make sure it's large enough. The default creates a node size that is smaller than the contour line width and won't show up. The segment node size example below is set to 7, and when paired with the settings in the figure above, the segment ends will show.



Finally, add the commands necessary to connect all the segments into a mesh. Ingen uses four segmented contours to create a mesh block, so 4 different segments need to be specified to form the four sides of a

block. However, specifying the mesh block as a square will constrain the problem to require 2 specified segments:

```
squareRule=altair.squareDistrib()

blk.metal_hemisphere = altair.block2(jMin = seg.inner_metal, jMax = seg.metal_he_bdy, rule = squareRule, material = mat.cu)
blk.he = altair.block2(jMin = seg.metal_he_bdy, jMax = seg.outer_he, rule = squareRule, material = mat.HE)
```

altair.squareDistrib() specifies we want square-ish zones.

squareRule is a variable that makes the code shorter.

blk namespace. There are two.

Both use block2 commands that take the inner segmented contour and the outer segmented contour as the constraints for how the HE will be zoned up. The squareRule provides the additional constraints.

- **blk.metal_hemisphere** creates the block for the metal hemisphere.
- **blk.he** creates the block for the HE.

Finally, we need to assign the blocks to a material. The result of the 3 commands looks like the mesh in figure 9.

Two more things to close this out:

```
seg.inner_metal.tag('pdv')

ingen.endModel(metadata=True,x3d=True,npes=144, matsAsReg = True)
```

seg.inner_metal contains the list of points the PDV will monitor, so it should be tagged.

- **('pdv')** tags the boundary to be used for PDV. This is used in the FLAG input to specify the boundary the PDV is monitoring.

ingen.endModel closes out the ingen session. If you run ingen on the file before the endModel, ingen will produce an error.

- **x3d=True** tells it to write the mesh.
- **npes=144** tells it to use up to 144 processors. Writing these x3d files takes a bit, so specifying way more processors that you will ever use is a waste of time. However, FLAG will crash if you don't create enough x3d files.

Running the python script using: ingen -g Cagliostro_Mesh.py, should create a mesh directory that contains mesh files and the PDV boundary points file. It will also return the part masses of the HE and the copper. The python script used to generate the mesh is included in the 'Files' section below. There is also a tar file of the mesh for those which avoids having to open Ingen, which is slow under certain circumstances (such as remoting in from home).

In the next chapter, we will write the FLAG input file to run this mesh.

Files

Python script file to generate the mesh: Cagliostro_Mesh.py
Mesh files in case you don't want to generate them: Cagliostro_Mesh_Files.tar.gz

8. Cagliostro: Simplified Mesh

Chapter 7 briefly overviews the Cagliostro experiment, described in LA-UR-87-2292..

This 4-part Cagliostro chapter adds to the HE cylinder FLAG input used in Chapter 6 and sets up a basic mesh in FLAG, results shown in figure 12. The FLAG input in this first section will be used in the rest of the Cagliostro section. To follow the instructions in this chapter, you can modify the cylinder file you created in Chapter 6, or you can download the file at the end of this chapter.

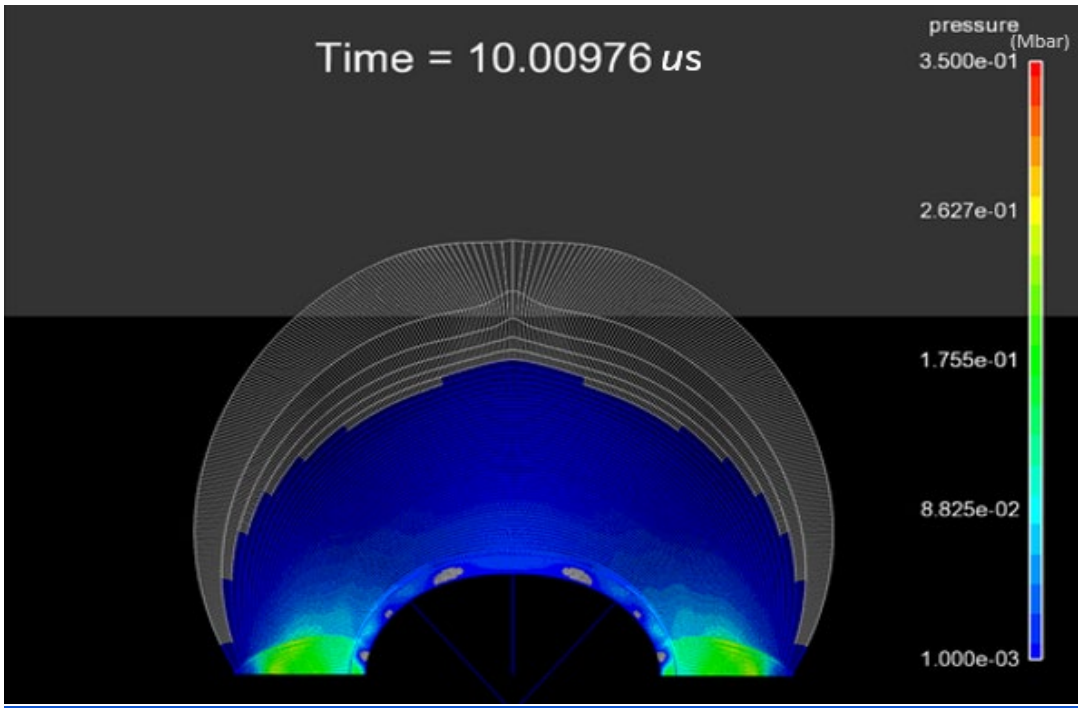


figure 12. This is a video, linked here , or you can click on the image

For Part 1, use the Ingen mesh files you made in Chapter 7, or download the Python file and mesh file (originally lined in the files section of Chapter 7).

The major difference between the input for the Cagliostro experiment and the cylinder input is that we are going to import an existing mesh to use instead of creating the mesh in the input. We also need to import the PDV boundary we created in Python.

Importing and Partitioning the Mesh

We need to pull in the mesh, then partition it. This is done in FLAG by importing the Ingen mesh called **mesh(donor)**. Then, a 2nd mesh will be used to take the donor mesh and partition it into equal size submeshes, which are sent to the various processors. Keep in mind that when you import an ingen mesh, you actually need to create 2 meshes. Both meshes need to have the geometry specified for them. The importing and partitioning is shown below:

```
mk /global/mesh(donor)/geometry/axis2
mk /global/mesh(donor)/zoner/importx3d
   filepath = "./mesh"
   file = "Cagliostro.x3d"

mk /global/mesh/geometry/axis2
mk /global/mesh/zoner/repartition
   meshname = "donor"
mk /global/mesh/zoner/repartition/partitioner/rcb
   meshname = "donor"
```

mesh(donor)

/geometry/axis2 for the donor mesh (cylindrical symmetry).

importx3d imports the mesh that we created.

The second set of commands creates a new mesh by partitioning mesh(donor) such that it most efficiently uses the available processors. This second unnamed mesh is the one that FLAG will actually use for the hydro.

Importing the PDV Boundary

The functions used for boundaries and the boundary creation commands are similar to those in the cylinder problem. The one notable exception is that we import the PDV boundary.

```
mk /global/mesh/kbdy(bdy_PDV)/importdefn
   filepath = "./mesh"
   file     = "Cagliostro.PDV.Bdy"
   nfields  = 21
```

bdy_PDV imports the PDV boundary created in ingen and assigns it to this boundary.
importdefn finds the files.

The hydro boundaries for this problem were simplified from the cylinder. We are once again using cylindrical geometry, so we need to set boundary conditions for the r-direction and z-direction. Here, We only needed boundary conditions for r=0 and for z=0, so there are just 2. We don't need boundaries for rmax and zmax because we will let them expand as the model evolves.

Importing Regions Created in Ingen

```
mk /global/mesh/kregion(reg_Cu)/importdefn
    filepath = "./mesh"
    file = "Cagliostro.copper.Reg"

mk /global/mesh/kregion(reg_HE)/importdefn
    filepath = "./mesh"
    file = "Cagliostro.HE.Reg"
```

importdefn is used to import the copper and HE regions**,** instead of creating them.

Recall these were created in Ingen in chapter 7.

reg_Cu and **reg_HE** are available for import in the /mesh directory.

Because these are the same names that we used in the cylinder problem, we can reuse the material blocks for both materials without making any changes.

The PDV block is essentially the same as for the HE cylinder, but we have to change the origin and directions to match the PDV directions in the experiment.

```
mk /global/mesh/output/visar_pdv
   bdy = "bdy_PDV"
   matlist = "mat_Cu"
   kkvll = 2
   vorigin = 0.0       0.0     0.0 &
             0.0       0.0     -0.943

   vdir    = 0.0       0.0     1.0 &
             0.643     0.0      0.766
```

bdy_PDV is the imported PDV boundary we are monitoring.

kkvll is the number of PDV probes we are putting into the model. In this case -- two PDVs.

vorigin PDV origin.

Note that we moved the 2nd PDV origin down in z to match the experiment. Also, note that in PDV coordinates, we are actually using the z location for the PDV z coordinate instead of using the y-location like we do with boundaries.

vdir is the normalized unit vectors of the PDV direction.

For probe 2 (the 50° probe), they are simply the Sin(50) and Cos(50) .

Finally, the light point needs to modified (not shown here, look at the .flg input file) to light the main charge at the top of it. Using 5.07cm puts it just inside the outer HE boundary.

Files

FLAG input deck: Cagliostro.flg
Dominic Cagliostro's experiment writeup: Cagliostro_Paper_1988_1.pdf
Digitized PDV data files: cagliostro_data.tar.gz

9. Cagliostro: Complex Mesh Parts and ALE Intro.

This section focuses on modifying the geometry to better match the original experiment. The unconfined booster glued to the top of the spherical charge is an especially difficult meshing issue. When the booster goes off, the zones reach pressures on the order of 0.3 Mbar. The air zones just behind the booster are at 1e-6 Mbar. That pressure differential really wants to squash the air zones. Additionally, the booster is expanding sideways at the booster/HE interface. The main HE wants to expand radially outward, which adds in all kinds of weird shear in the zones adjacent to the booster/HE interface.

HE free expansion into air is most easily dealt with using a full Euler treatment. However, we will walk through running the problem with ALE. ALE settings and setup have to be just right to get it to run to completion. See figure 13 for the resulting video. Minor changes to this geometry, say even a FLAG version change (I used 3.8.Alpha.19 here), will likely result in a problem that doesn't complete.

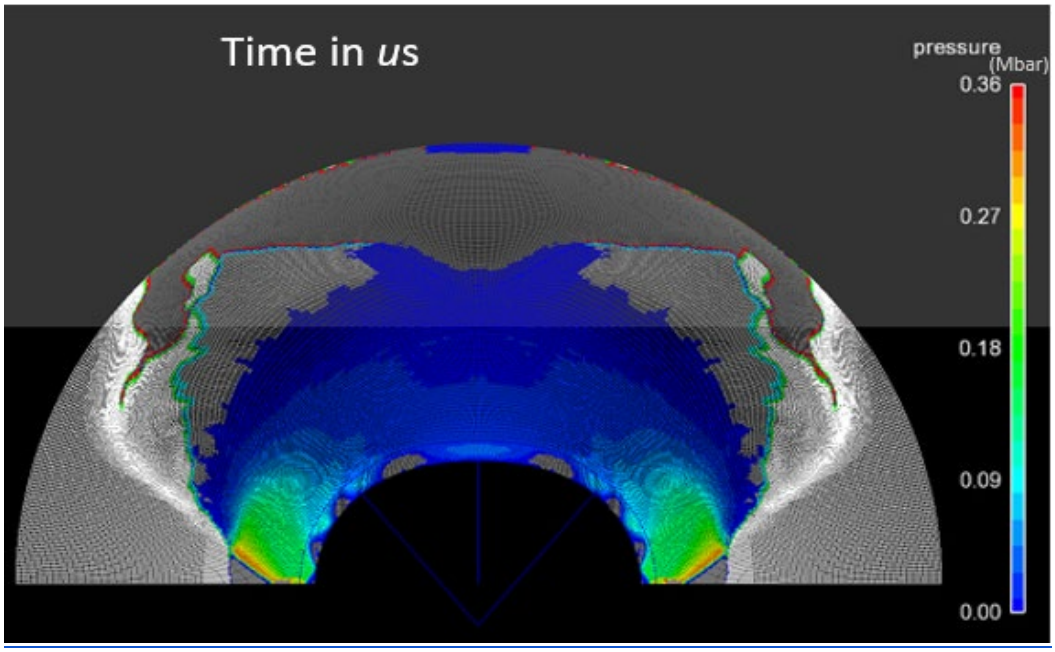


figure 13. This is a video, linked here , or click on the image

Modifications to the Python Mesh Script

Make some modifications to the Ingen python script in order to add an air region outside of the HE:

1. 'paint' a booster onto the air region
2. remove some of the outer diameter of the HE along the edge to better match what was fired in the Cagliostro experiment
3. 'paint' air onto the HE to represent the removed HE (results shown in figure 14)

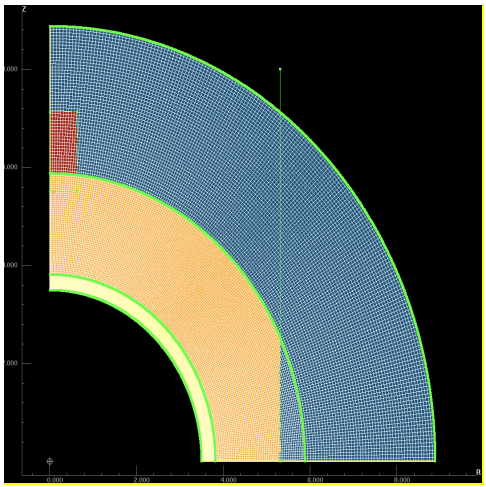


figure 14.

Conforming versus Painting Mesh

To understand how painting works, take a look at the booster in figure 14 (in dark red). The outline of the booster region is the green surrounding line (extending into the main HE, which will be discussed more later). Paint works by turning any of the underlying mesh inside of the booster region to "booster material". Similarly, looking at the HE cutout on the sides, paint looks at the HE material lying inside the cutout region and turns it into air. Note that material boundaries are not perfect. There is some raggedness to them as they follow the underlying mesh zones. However, paint can insert complex shapes into conformal meshes with relative ease.

When to paint and when to conformally mesh? Conformal meshing is recommended when it can be done quickly, painting is recommended when the effort to conformally mesh a part is great. For example, if the parts were U.S. states, I recommend you conformally mesh Colorado because it's a nice 4 sided figure, but paint Michigan. Conformal meshing is also recommended when the part is close to a diagnostic of interest. In this problem, the inner metal surface is conformally meshed because the PDV is looking at the inner metal surface. This avoids paint errors. However, if the item/surface is on the other side of the problem from the diagnostics, then painting it is recommended. Painting is fast and something on the other side of the problem probably won't have time to create significant errors. In this problem, the booster and the HE cutout are painted because they are on the far side of the problem from our PDV.

Python Mesh File Modifications

Modify the simple Cagliostro_Mesh.py created in Chapter 7, or download the final mesh file.

- First, in the materials section, add an additional material for the booster called **mat.booster**.

```
cntr.cut = gwiz.rLine(r = he_truncation, z1 = 0, z2 = 8)
cntr.air = gwiz.offset(cntr1 = cntr.outer_he, dist = 4.0)
cntr.booster = gwiz.loadTable('booster.rz')
```

- Then, add in some additional contours:
  cntr.cut is going to represent the cutout location on the HE.
  This line is for visual purposes to see where I'm going to paint in air on the HE. A little later, I'm going to define a box that I will actually use for paint purposes. That box is 3D and doesn't show up in our model. So I stuck a line in there so I could see the paint boundary.
  cntr.air Offset the outer HE contour in R by 4.0 cm. Use this new contour as an air boundary.
  cntr.booster Load an rz table that represents the outline for the booster.

Figure 15 shows what the 3 additional contours look like before mesh is added to the air region. Note the booster contour. The HE boundary is extended into the HE, which will ensure the contour intersects the HE outer boundary across the entire radius, even though the exact HE boundary data aren't available. It's fine that the booster contour extends into the HE because that region will not be painted on.

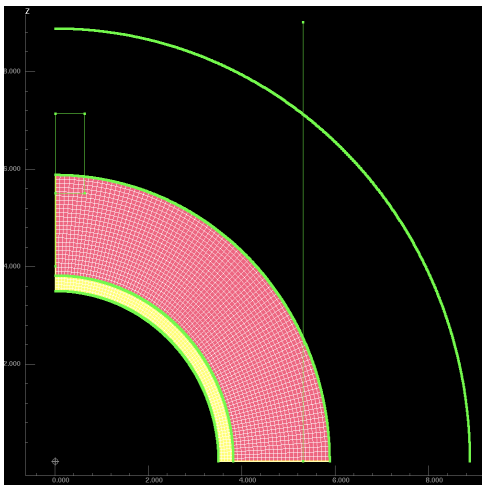


figure 15. Note the booster contour extending into the HE

Figure 16 shows the air mesh and region added in:

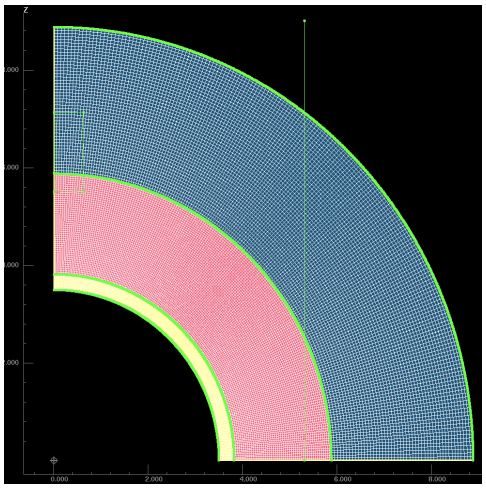


figure 16.

Now that we have enough mesh, we can start painting onto the mesh. This is done with the following lines of code :

```
gwiz.solid.contours2Regions(contours = cntr, regions = reg)
blk.air.paintFeature(osoRegion = reg.booster, baseMaterial = mat.air, featureMaterial = mat.booster)
reg.cutout = gwiz.solid.box( min = (he_truncation,0,0), max = (7,7,0))
blk.he.paintFeature(osoRegion = reg.cutout, baseMaterial = mat.HE, featureMaterial = mat.air)
```

contours2Regions command on our cntr namelist converts contours to regions.
blk.air.paintFeature is a function that paints on the air block (blk.air).
Even though the booster contour extends into the HE, none of it gets painted because that's a different material block. To paint on the HE, use a **blk.he.paintFeature** call.
osoRegion uses Oso functionality to paint onto the mesh. OSO needs regions in order to paint.
baseMaterial is the material we want to paint on (examples: air or HE).
featureMaterial is the material we want the painted regions to be.
In this case, we want painted regions to be booster.
gwiz.solid.box region command creates a 3D box that produces an HE cutout on the edge.
Everything inside the box is painted as air. However, ingen does not display 3D objects on a 2D mesh such as this one, so the box is not shown. That's why I created cntr.cut -- so I could see where the edge of the box was!

The final thing to do is tag some of the interface boundaries. These will be used for some ALE boundaries in the FLAG input later. Those final commands look like:

```
seg.metal_he_bdy.tag('metal_he')
seg.air.tag('air')
```

Run the python script to generate the mesh, or download this meshfile and booster.rz file. The booster.rz file needs to be in the same directory as the python mesh creation file.

FLAG Input Modifications

Figure 17 shows a simulation with no ALE. If the input is modified to use the new materials, with no mesh controllers, the mesh will tangle within the first 0.9 μs because the booster compresses the air region just on the back side of it.

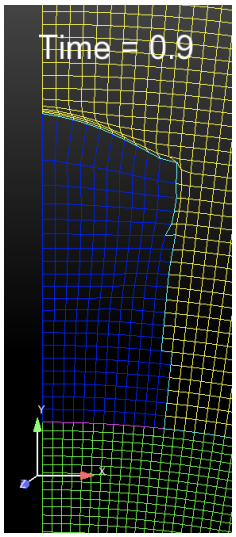


figure 17.

Adding the air made it easy to paint on a booster, but it squashed the mesh zones. In order to deal with this, we will have to add in mesh controllers, colloquially known as ALE (Arbitrary Lagrange Eulerian).

A note about ALE functionality: all the points along the ALE boundary need to be told how to behave. If you have points along an ALE boundary that are not subject to a boundary condition, then FLAG will crash. The ALE region used here is bound by the following 4 boundaries: metal_HE boundary, rmin boundary, zmin boundary, air boundary.

We tagged the metal_HE boundary and the air boundary in the Ingen file. We will define the rmin and zmin boundaries in the .flg input in a bit.

Next, modify the FLAG input from the Cagliostro Part 1. I will only comment on those modifications.

FLAG Boundary Imports

bdy_rmin and bdy_zmin are already defined; metal_HE boundary and the air boundary are still needed. Use **importdefn** to import the metal_HE boundary and the air boundary tagged in the meshing script to the FLAG input script. This is the same method used with the PDV boundary in the previous section on simplified mesh PDV boundary.

FLAG Boundary Condition Assignment

Next, assign some boundary conditions to the outer air boundary. Below is a boundary condition that does not allow the boundary to move:

```
mk +kbc(bc_outer)/kfix
   bdy = "bdy_air"
   nfix = 1 1
```

Fixing the boundary causes the sweeping shock/pressure wave from the detonation to create a large pressure reflection and the mesh to pile up on the boundary. The weird vortices at the edge of the mesh ultimately tangle. However, the boundary does not have to be fixed. Changing it to nfix = 0 0, allows the boundary to expand outward. I found that it was easier to fix the boundary and deal with the pressure and mesh piling up on the boundary.

Import the New Regions

The newly created booster region and air region need to be pulled into the input:

```
mk /global/mesh/kregion(reg_air)/importdefn
        filepath = "./mesh"
        file = "Cagliostro.air.Reg"
        nfields = 21

mk /global/mesh/kregion(reg_booster)/importdefn
        filepath = "./mesh"
        file = "Cagliostro.booster.Reg"
        nfields = 22
```

Our First ALE Block

ALE is a powerful meshing tool, but takes trial and error to figure out. This chapter covers the basics of properly implemented ALE strategy.

First, put in the capability to advect material across zones:

```
mk /global/mesh/interface/volf

mk +priority/matlistpty
   matlist = "mat_Cu" "mat_HE" "mat_air"

mk /global/mesh/optimize/ale/advection
   iuse_legacy_ale = 0
   errcrit1 = 1e-4
   iedmfmeth = 2
```

volf is an interface tracking and reconstruction command.
matlistpty is a statement which controls material priority.
    We specify the material by name, the order of the names is the priority.
advection functionality must be invoked.
    The advection variables listed directly below the advection command are the ones suggested by FLAG developers.

Next, add in a relaxer.

```
mk /global/mesh/optimize/ale/volrelax/cn
        regions = "reg_HE" "reg_booster" "reg_air"
        relaxtime = 0.0
        relaxstop = 100.0

mk +kbc(bc_rmin)/kfix
        bdy = "bdy_rmin"
        nfix = 1 0

mk +kbc(bc_zmin)/kfix
        bdy = "bdy_zmin"
        nfix = 0 1

mk +kbc(bc_metal_he)/kfix
        bdy = "bdy_metal_HE"
        nfix = 1 1

mk +kbc(bc_air)/kfix
        bdy = "bdy_air"
        nfix = 1 1
```

cn stands for a mesh relaxer type called condnum.

There are other options, like adapt. There is an adapt code block that is commented out in the .flg file included in this chapter. Swap the cn section for the adapt section to see the difference.

regions = "reg_HE" "reg_booster" "reg_air" tell the relaxer what regions will be relaxed.

The main HE charge, booster, and air region are specified - the relaxer will be applied to them.

relaxtime and **relaxstop** tell when the controller turns on and when to stop it.

I want the controller to be active throughout this entire problem.

mk +kbc are four different boundary fix conditions for ALE. Very similar to hydro boundary conditions.

'+' kbc stands for **/global/mesh/optimize/ale/volrelax/cn/kbc/kfix**

This '+' does ***not*** stand for the hydro boundary conditions found under the node **/global/mesh/kbc/kfix**. We need TWO boundary conditions in an ALE implementation -- hydro boundary conditions and ALE boundary conditions. The ALE boundary conditions are different and independent of the hydro boundary conditions. The ALE boundary conditions are needed because the boundary points are being moved around due to geometric issues and not due to hydrodynamic features like pressure. We need a different set of boundary controllers to tell ALE how those points should move. The rmin boundary kfix condition fixes points in r but allows motion in z. Thus along the rmin axis, ALE can move points in the z-direction but cannot move points in the r-direction. As mentioned above, hydro and ALE kfix conditions are independent of each other. It's possible to allow points to move under hydro and fix them under ALE, or vice versa. Note: for each type of ALE implemented, each controller will need to have boundary conditions for THAT relaxer. For example, if 5 different ALE relaxers were used, boundary conditions will be needed under each one to tell the relaxers how to handle the boundary.

Finally, we are going to add in a geometry controller that will limit the portion of the mesh that undergoes ALE relaxation:

```
mk +controller(where)/geom
   enable_ang = 1
   ang = 45.0
   enable_elength = 1
   elength = 10e-3
   enable_sratio = 0
   enable_eratio = 1
   eratio = 4
   enable_dcn = 0

   izaway = 5
```

enable_ang=1 turns the angle check in the geom comptroller.
This will ALE any zone whose angle between two vertices is less than 45°.
enable_elength=1 enables the edge length check.
This will ALE any zone whose edge length drops below 10e-3.
enable_eratio =1 enables edge length ratio check.
This will ALE any zone whose ratio of edge lengths is greater than 4.
izway=5 helps when a zone undergoes ALE by using zones up to izaway to fix the distorted zone.

Run the FLAG input. This is a tough problem to run ALE. Things to try if it doesn't complete:

1. Modify the geom options.
2. Try a different relaxer type.
3. Increase or decrease the air boundary offset that's found in the cntr.air part of the mesh script.

Comparison of Two Models

This section compares this model with the simplified mesh from Cagliostro Part 1.

In Cagliostro Part 1, the problem was simplified by leaving the booster out and not truncating the edges of the HE. Adding in those parts created quite a bit more work. Is adding those experimental details worth the effort?

A comparison of the PDV probes from the simplified geometry (Cagliostro Part 1) and from this one (Cagliostro Part 2) is shown in figure 18.

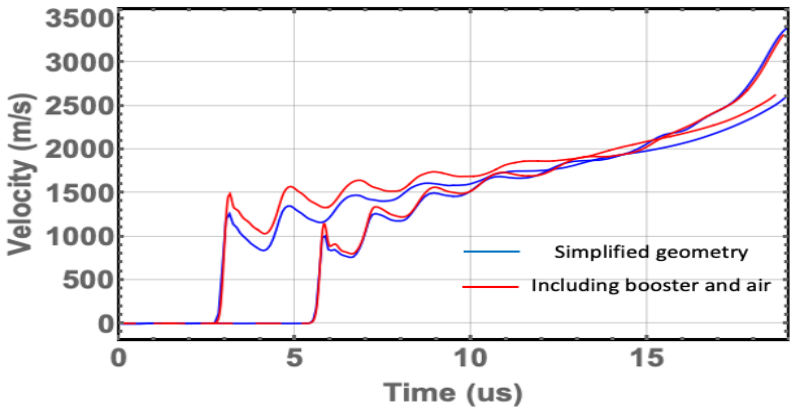


figure 18.

The first probe to jump off is the 0° probe directly under the booster. The second probe to jump off is the one at 50°. The 0° probe clearly shows significantly different velocity (about 200 m/s) when the booster is included. The 50° probe shows minor differences between the two treatments. Therefore, it is worth the trouble of including the booster if an error on the order of ~200m/s is not tolerable.

To parse the PDV data (still called *.visar by FLAG), use the following simple Python script to spit out individual probe files in a comma-delimited format. To use it, load python 3.6 and execute at the prompt:
python FLAG_PDV_parse.pv Cagliostro.visar

Files

Python script used to generate the mesh: Cagliostro_Mesh_ALE.py
Booster r-z file: booster.rz
Mesh: Cagliostro_ALE_Mesh.tar.gz
FLAG input script: Cagliostro_ALE.flg
FLAG PDV parser: FLAG_PDV_parse.py

10. Cagliostro: Running Eulerian in FLAG

This section walks through some modifications to the previous FLAG input file and some minor changes to the meshing script, with the goal of running the Cagliostro problem using an Eulerian approach. Figure 19 links to a movie that shows our previous Lagrangian/ALE model on the left side and the full Eulerian model on the right side.

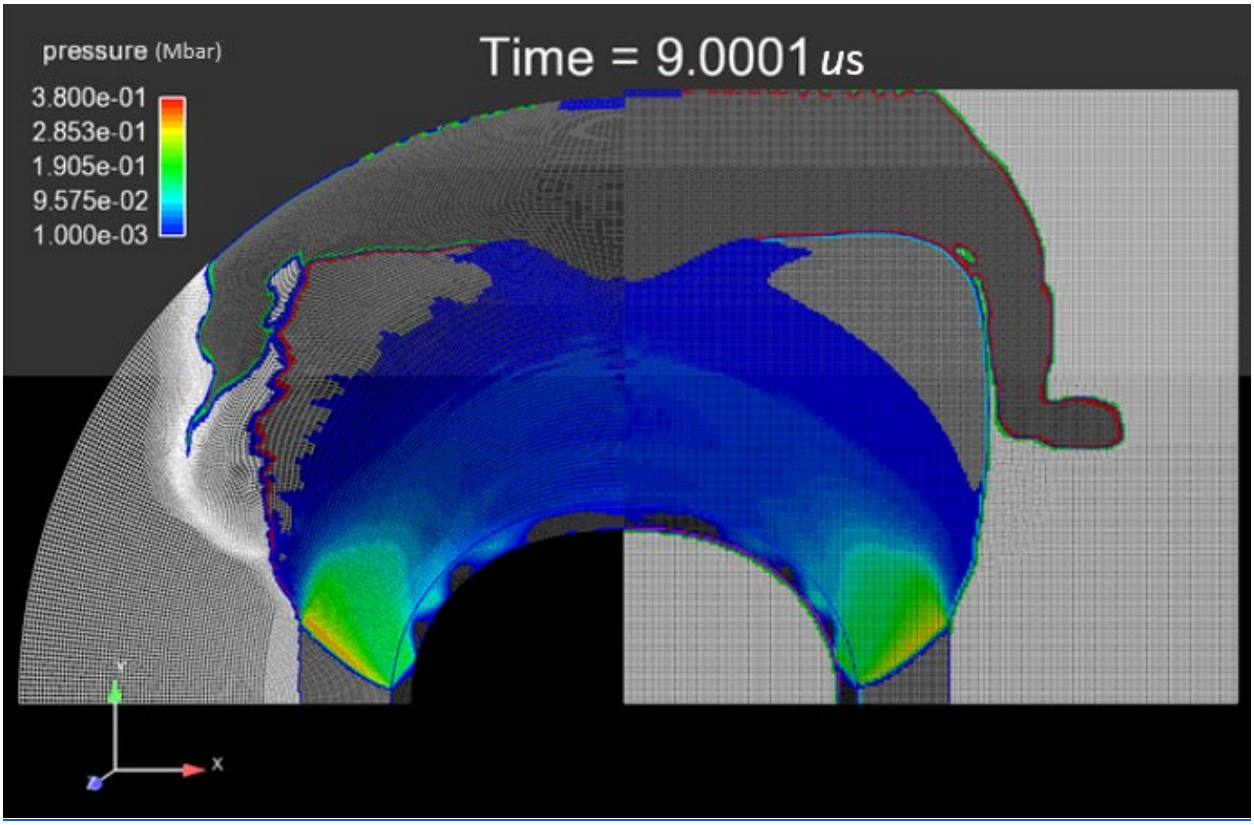


figure 19. This is a video, linked here , or click on the image to open. Langrangian on left, Eulerian on right.

Note how the mesh moves with the hydrodynamics in the Langrangian model, but is fixed with materials flowing through the mesh in the Eulerian model. When working with freely expanding HE products, Eulerian is a simpler approach.

Modifications to the Meshing Python Script

To run an Eulerian model, the recommended procedure is to build geometry in Osito, then use OSO files to load the geometry into an Eulerian model. Osito is much easier to use than Ingen. Alternatively, region files from Ingen input can be exported as OSO files. For the example in this chapter, a good Ingen meshing script is available. Adding a few lines to that input will produce the needed OSO regions.

```
altair.blocks2Regions(blocks=blk,regions=reg)
reg2 = gwiz.solid.makeRegions()
reg2.reg_HE = reg.he
reg2.reg_air = reg.air
reg2.reg_booster = reg.booster
reg2.reg_metal = reg.metal_hemisphere
gwiz.writeOso(reg2,'Cagliostro.oso')

ingen.endModel(metadata=True,x3d=False,npes=144, matsAsRegs = True)
```

altair.blocks2Regions turn material blocks into regions that live in the *reg* namespace.
gwiz.solid.makeRegions creates a new region namespace called *reg*2.

Our OSO file needs metal, HE, and booster regions. Currently, *reg* contains those regions, and a number of other regions which will not be used. So I will copy only the regions I want to use over to *reg2*.

gwiz.writeOSO saves the OSO file.
endModel command, set **x3d = False** so mesh files aren't written, as they won't be used.

Execute the script (included at the end of this section). This will produce a Cagliostro.oso file that contains your geometry.

Modifications to FLAG Input

First, use **pszoner** to make a rectilinear mesh:

```
mk /global/mesh/geometry/axis2
mk /global/mesh/zoner/pszoner
   use_interval = "yes"

   ranges(1,:) = 0.0 10      !r dimensions
   izones(1,:) =    200      !Number of r zones (0.05 cm zones)

   ranges(2,:) = 0.0 10      !z dimensions
   izones(2,:) =   200       !Number of z zones (0.05 cm zones)
```

To reuse the previous Cagliostro ALE input file:

- Remove the PDV, air, and Cu_HE boundary import statements. Importing boundaries won't be necessary, and trying to import them will cause FLAG to crash.
- Add in appropriate functions and boundary conditions for rmax and zmax.
- Add in hydro boundary conditions to keep the outer mesh boundary fixed.

```
mk /global/mesh/hydro/lhydro
   alias pressure zp
   alias density zr
   alias velocity pu
   cflu = 0.7
   cflv = 0.2

mk +kbc(bc_zmin)/kfix
   bdy = "bdy_zmin"
   nfix = 1 1

mk +kbc(bc_zmax)/kfix
   bdy = "bdy_zmax"
   nfix = 1 1

mk +kbc(bc_rmin)/kfix
   bdy = "bdy_rmin"
   nfix = 1 1

mk +kbc(bc_rmax)/kfix
   bdy = "bdy_rmax"
   nfix = 1 1

mk +mixedcell/mxtip
   alpha = 1.0
   facm  = 0.0
   pfloor = 1e-12
```

- Add the recommended values, inserted above, to lhydro and to mxtip.
- Note that cflu and cflv are very large compared to Eulerian settings on the LAP page.

In the next section of code, regions are imported from the OSO file:

```
mk /global/mesh/kregion(universe)/universe

mk /global/mesh/kregion(reg_Cu)/oso_paint
        oso_fnames  = "Cagliostro.oso"
        oso_regions = "reg_metal"

mk /global/mesh/kregion(reg_HE)/oso_paint
        oso_fnames  = "Cagliostro.oso"
        oso_regions = "reg_HE"

mk /global/mesh/kregion(reg_booster)/oso_paint
        oso_fnames  = "Cagliostro.oso"
        oso_regions = "reg_booster"

mk /global/mesh/kregion(reg_air)/oso_paint
        oso_comp = "implicit"
```

- The final region the air will occupy is created by filling in all the zones that are not currently filled with a material. This is similar to the 'void' region in some codes. The **oso_comp = "implicit"** creates a region of all the unfilled zones called reg_air

```
mk /global/mesh/output/visar_pdv_ale
   matlist = "mat_Cu"
   kkvll = 2
   vorigin = 0.0       0.0     0.0 &
             0.0       0.0     -0.943

   vdir    = 0.0       0.0     1.0 &
             0.643     0.0      0.766

   doc BuffVisarPDV every 10
   dot VisarPDVDump every 1.0
```

- Add **visar_pdv_ale**, which is a PDV treatment that can deal with advection.
  It is very similar to visar_pdv, except it doesn't need a boundary specified. Specify the material and it will follow the surface of the material. This is the last change to make.

The PDV results from the Eulerian model and from the more complex Lagrange/ALE model are plotted in figure 20. The Eulerian PDV has a high frequency oscillation, likely due to the PDV probe sampling different zones as a function of time. The Eulerian also has a longer rise time. Otherwise, the two models give similar PDV results.

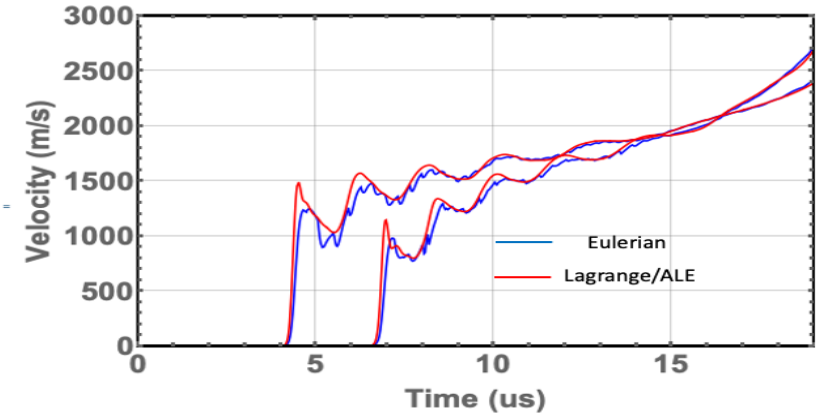


figure 20.

Files

FLAG input : Cagliostro_Euler.flg
Python script used to generate OSO file: Cagliostro_Mesh_Euler.py
OSO geometry file: Cagliostro.oso

11. Cagliostro: AMR Implementation

In figure 21, compare the original Eulerian version using a uniform mesh on the left with the AMR implementation on the right.

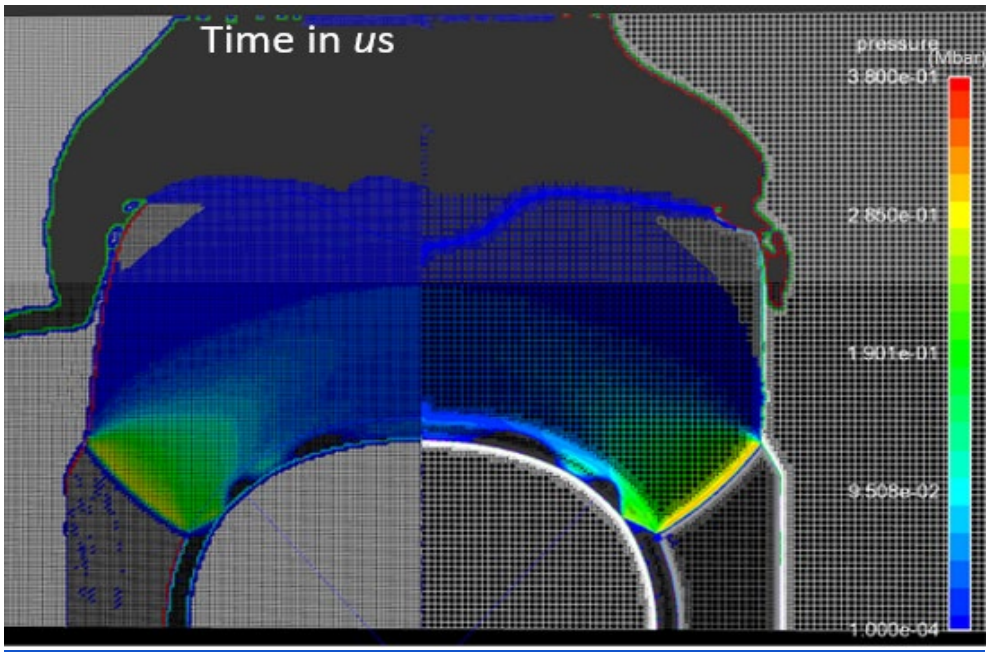


figure 21. This is a video, linked here , or click on the image above to open the movie. Eulerian with uniform mesh on the left, AMR on the right.

Figure 22 zooms in closer to view the mesh evolution as the detonation breaks out into the main charge. Notice the AMR model has a coarser grid compared to the non AMR. However, the AMR model can resolve down to a factor of 4X higher resolution in regions that need higher resolution. Yet, both of these models took about the same amount of time to run.

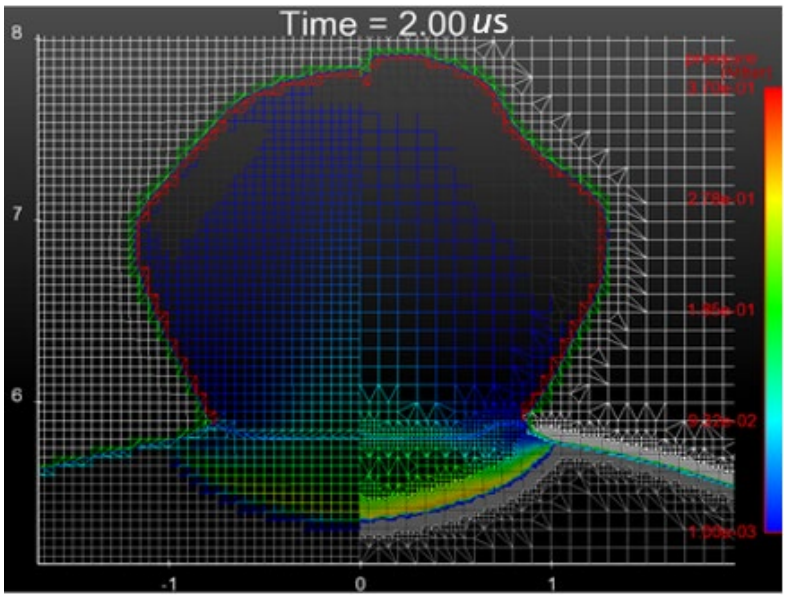


figure 22. Fixed resolution mesh on the left. AMR mesh on the right.

The reason for AMR: it allows for high resolution where it's needed and allows for extremely coarse resolution where it's not needed. Probably the most obvious example of this is the HE. Within ~+/- 1 mm of the detonation front, resolutions down to 10 µm are needed. Once beyond the detonation region, 1mm zones provide a good representation of the long-term HE energy delivery. For this problem, if we assume the HE detonates at 8mm/us, the HE detonation covers a 1 mm distance in 125 ns. We need high resolution in that 1mm region for 125 ns. After that, the remaining 19.875us of this problem, that 1mm can be run with coarse zones (e.g 1mm zone sizes). AMR allows the flexibility to use high resolution only when and where it is needed and use coarse resolution otherwise.

Modifying the FLAG Input to Add in AMR

This section will detail modification of the input from Cagliostro Part 3 and will reuse the Cagliostro.oso geometry file. Or, download the full FLAG input.

- First, define a variable to allow for easy changes to the number of refinement levels. Otherwise, changes to the initial meshing are almost identical to Part 3, except that the **mesh(donor)** needs to be changed (see below).

```
integer amr_level
amr_level = 2

mk /global
   title = "Cagliostro"
   tstop = 20.0
   dtinitial = 1.0e-4

mk /global/mesh
    volfractol = 1e-8
    cfl = 0.7
    izcen = 2

mk /global/mesh(donor)/geometry/axis2
mk /global/mesh(donor)/zoner/pszoner
   use_interval = "yes"

   ranges(1,:) = 0.0 10      !r dimensions
   izones(1,:) =   100       !Number of r zones (0.1 cm zones)

   ranges(2,:) = 0.0 10      !z dimensions
   izones(2,:) =   100       !Number of z zones (0.1 cm zones)
```

- Next, add the acceptor mesh, which will determine the amount of refinement to put on the mesh at the start of the problem.
- Add a function to deal with the split ends that AMR adds to the problem.

```
mk /global/mesh/geometry/axis2
mk /global/mesh/zoner/refine_donor
        meshname = "donor"

mk +where(reg_Cu)/oso_region
        oso_fnames  = "Cagliostro.oso"
        oso_regions = "reg_metal"
        level_reg = 0
        level_bdy = amr_level

mk +where(reg_HE)/oso_region
        oso_fnames = "Cagliostro.oso"
        oso_regions = "reg_HE"
        level_reg = 0
        level_bdy = amr_level

mk +where(reg_booster)/oso_region
        oso_fnames = "Cagliostro.oso"
        oso_regions = "reg_booster"
        level_reg = 0
        level_bdy = amr_level

mk /global/mesh/splitend
        ignore_hydro = 1
```

- Make changes to a few default settings under lhydro and advection. The rest of the input is mostly unchanged.
- The last remaining significant change is putting in the AMR block. ***Note that the AMR block must come after the material definitions***. Currently, material assignments must be before AMR so that **matregion** knows what the materials are.

The AMR block looks like:

```
mk /global/mesh/optimize/amr2/isotropic
    iremap_order = 2
    islopelimiter = 1

mk +controller(C1)/matregion
    matlist = "mat_Cu" "mat_HE" "mat_booster" "mat_air"
    level_internal_bdy = amr_level amr_level   1             0
    tstart = 0.0
    tstop = 99.0

mk +controller(zr)/smoothmonitor
    var_monitor = "density"
    level = amr_level
    ismooth_method = 2
    iter_smooth = 10
    alpha = 0.5
    beta = 1.0
    tstart = 0.0
    tstop = 99.0

  mk +controller(zp)/smoothmonitor
    var_monitor = "pressure"
    level = amr_level
    ismooth_method = 2
    iter_smooth = 10
    alpha = 0.5
    beta = 1.0
    tstart = 0.0
    tstop = 99.0
```

There are three mesh controllers, the second two controllers refine based on density gradients and pressure gradients.

matregion mesh controller is based on refining the materials at the material boundaries.

Files

FLAG input: Cagliostro_Euler_AMR.flg
Geometry file: Cagliostro.oso

Acknowledgements

This practical introduction to FLAG is based on a series of online tutorials created by the author while telecommuting. We'd like to acknowledge Dominic Cagliostro, Scott Jackson and Eric Anderson for sharing experimental data. Devin Shunk, Erik Shores and other confluence commenters for encouraging this work and improving the online tutorial. Jim Hill for the FLAG manual. Nick Denissen for his tutorials and for answering numerous questions from the authors.